Description of a Telemetry Procedural Language

R. I. Ścibor-Marchocki
Flight Operations and DSN Programming Section

A procedural language and a compiler for it are being developed as an aid in the writing of programs which will process telemetry data received from spacecraft. This article describes the language. Also, the philosophy that leads to the choice of the language is briefly presented.

I. Introduction

A study has been undertaken to demonstrate the effectiveness of a high-level programming language in the design of a mission-independent telemetry processor. While an assembly language provides greater freedom to the programmer, the greater ease, hence speed, with which a program can be written in a high-level language can be decisive in a real-time environment.

As a first step in this study a Telemetry Procedural Language (TPL) has been defined. TPL is oriented toward the specific programming procedures involved in a subset of a telemetry processor. This subset has been chosen because (1) it constitutes a relevant part of the whole telemetry software subsystem, and (2) it requires extensive changes from one mission to another. In the design of TPL, a large choice of sophistication nuances was available. A descriptive approach requires from the programmer only the description of the processor. A more functional approach provides the programmer with very powerful tools, but the concern with the program logic is left to him. Each approach has its own advantages as well as disadvantages. A descriptive language depends heavily upon the actual configuration of the telemetry processor, and the corresponding compiler generates progressively worse code as the configuration changes from the original one. A functional language, however, is much more versatile and more suitable in a mission-independent environment. Hence, TPL has been defined as a functional language.

II. Formal Specification of TPL

A given language, e.g., TPL, has to be described in some language. A language which is used to describe another is called a meta-language. English can serve as such a meta-language. However, ordinary English is not concise enough and lacks the necessary precision. The Backus–Naur Form (BNF) is somewhat better, but still inadequate.

The transmogrification language (TMGL) is similar to the BNF and may be used to specify a given language. TMGL is both human-readable (i.e., a publication language) and computer-readable [i.e., by the transmogrification (TMG) system]. A specification of a given language, consisting of the grammar and a translation into another language (usually an assembly language) written in TMGL, can be compiled by the TMG system to yield a compiler for the given language. Therefore, such a speci-

fication of a given language is definitive: There can be no discrepancy between the human-readable specification and the compiler's performance.

This article provides an informal specification of TPL. A formal precise specification of TPL is being written in TMGL.

III. The Features of TPL

TPL is a FORTRAN-like language. Most of the usual features of FORTRAN IV H are provided in TPL. The four features of FORTRAN IV which intentionally are not part of TPL are:

(1) There are *no* DOUBLE PRECISION variables or constants.

(2) There are *no* COMPLEX variables.

(3) Variables may *not* be declared as either REAL or INTEGER. Instead, they are identified exclusively by their initial letter.

(4) The EQUIVALENCE statement is *not* allowed.

TPL is intended to be a living, growing language; thus, new features will be introduced as the need for them arises. The following material describes those features, beyond FORTRAN, which already have been introduced into TPL.

A. Commas Are Optional

Commas are optional as syntactical delimiters between like entities. If a syntactical confusion would result, a delimiter is necessary but may be either a space or a comma. For example, one may declare

```
DIMENSION A(11 12 13)B(14 15)
```

The easy safe procedure that may be adopted by the programmer is to employ a space wherever FORTRAN would desire a comma or a space. However, the specific rules are as follows:

0 Excess spaces are permissible, except inside of a word.

1 A space has to be used as a delimiter between the logical operators X or V and any preceding or following name of a variable.

2 A space has to be used as a delimiter between the logical operators X or V and any following constant.

3 A space has to be used as a delimiter, on one side only, when the logical operator . is preceded by an integer constant and followed by a constant.

4 A space or a comma has to be used as a delimiter between a constant and a following constant.

5 A space or a comma has to be used as a delimiter between a variable and a following variable or constant.

6 A space has to be used as a delimiter between a keyword and a following variable or constant.

Violation of any of these rules may lead to an unpredictable misinterpretation, consisting of one or more of the following results, listed in order of decreasing probability:

1 Erroneous target code, which will yield incorrect computed results during execution.

2 The error message "incomprehensible."

3 Correct recovery.

4 Other error message(s).

5 If a subroutine or function call is involved, a failure during execution.

6 Partial recovery, which will yield correct computed results but be wasteful of computer time.

B. Active Field Is Columns 1–5, 7–72

A non-blank column 6 indicates that a given physical card is a continuation of the preceding physical card. Column 6 may *not* be used for any other purpose. The active field on any physical card consists of columns 1 through 5 inclusive and 7 through 72 inclusive. A logical card consists of the active field of any physical card with a blank in the continuation column and the active fields of as many continuation cards as contiguously follow the given physical card. Both the statement number, if present, and the statement may be placed anywhere on a logical card, subject to the constraint that the statement number, if present, must precede the statement on a given logical card.

C. Asterisk * for Comment

The asterisk * is used as a designator of a comment card. This asterisk must be the first non-blank character in the given logical card. It need not be in column one.

D. Mixed (IBM vs UNIVAC) Style Punching

TPL utilizes a 54-character set. Since the IBM 407 has only 48 positions on its print wheels, it cannot list cards of the TPL source language unambiguously. Forty-four characters are identical on the IBM 360 and UNIVAC 1108/9300 computers:

26 Alphabetic characters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
10 Numeric characters	0123456789
1 Space | blank	ƀ
7 Special characters	,$.-*/;

These characters may be keypunched or listed on any modern equipment without any difficulty. An additional 10 special characters differ among the various modern equipments:

plus	+
hyphen	−
right parenthesis	)
logical OR	|
ampersand	&
logical NOT	¬
apostrophe	'
double quote	"
equal sign	=
left parenthesis	(

For these characters TPL will accept either the IBM system 360 punching or the UNIVAC 1108/9300 punching, mixed in any manner intra | inter card. Hence, cards may be punched originally or corrected in either style, at the immediate convenience of the programmer, but with a view toward which equipment he intends to employ for the listing of the deck.

E. IF|OTHERWISE|ANYHOW

The blocking set of keywords

IF|OTHERWISE|ANYHOW

is available and preferable to the use of numbered statements for the execution of exclusive alternative blocks of coding. A complete description of this feature is contained in the articles referenced in the Bibliography and will not be repeated here.

F. DO|DO_END

The blocking set of keywords DO|DO_END is available and preferable to the use of numbered statements for the execution of a DO-loop. See the references for a complete description of this feature.

G. SECTION

The keyword SECTION can be employed for semantically delimiting local blocks of coding with regards to real names and statement numbers. Thus, the source coding within each numbered SECTION must use statement numbers in the closed integer interval [1, 999] and the same numbers may be reused in other sections. Similarly, the same source names may be reused in other sections. THE SECTIONs must be numbered in strictly increasing order with the numbers taken from the closed integer interval [1, 25].

H. UP|DOWN

The keywords UP|DOWN may be employed to nest blocks of local variables (but *not* statement numbers) to any depth. A new SECTION will force as many UP steps as necessary to return to the standard level. An automatic drop down to the previous level will *not* follow.

I. An Arithmetical Sub-atom Includes a Logical Sum in Parentheses

In addition to the two standard FORTRAN recursions resulting from the definition of a logical atom to include a logical sum in parentheses and an arithmetical sub-atom to include an optionally signed arithmetical sum in parentheses, the third recursion resulting from the definition of an arithmetical sub-atom to include a logical sum in parentheses is permitted. The BNF definitions are

$$\langle \text{LA} \rangle = (\langle \text{LS} \rangle) \mid \left\{ \begin{matrix} \& \\ \neg \end{matrix} \right\} \langle \text{LA} / \mid \langle \text{AS} \rangle \langle \text{relational operator} \rangle \langle \text{AS} \rangle \mid \langle \text{primitives} /$$

$$\langle \text{AAI} \rangle = (\langle \text{SAS} \rangle) \mid (\langle \text{LS} \rangle) \mid / \text{primitives} \rangle$$

J. NEW_DECK

For multiple compilation, the compiler program is reinitialized by the use of the keyword NEW_DECK. A new HEADER card optionally may follow immediately.

K. ASSIGN

The built-in in-line integer replacement function ASSIGN uses the syntax

$$\text{ASSIGN } \langle \text{scalar integer variable} \rangle = \left\{ \begin{matrix} \langle \text{integer constant} \rangle \\ \langle \text{arithmetic sum} \rangle \end{matrix} \right\}$$

Thus, the syntax is the same as that of an arithmetical equation, but preceded by the keyword ASSIGN. For example,

ASSIGN I = 3

L. Logical Operations Performed on 32 Bits in Parallel

The logical variables are 32 bit integers with

0 = .FALSE.

−1 = .TRUE.

The choice of −1, rather than +1, for TRUE yields an all-one-bits word as TRUE. Such a word easily may be masked to yield one-bits at any desired position in the word without the necessity of shifting or propagating operations. Each of the logical operations, as well as the statement

IF (⟨LS⟩) ⟨integer variable which reduces to a scalar⟩ = ⟨LS⟩

is executed as 32 bits in parallel subject to the same indicated operators. In all other logical IF statements; i.e.,

IF (⟨LS⟩) . . .

the argument of the IF is tested for non-zero; namely,

IF (⟨LS⟩ .NE. 0) . . .

In summary, .FALSE. is always zero. .TRUE. is generated as minus one, is computed upon as 32 bits in parallel, and is used as non-zero; i.e., at least one bit being a one.

M. Extracting Triplet

An extracting ordered triplet has been introduced as an additional logical atom. This triplet permits the unpacking of integer|logical variables and right justifying them in one operation. (This is similar to the UNIVAC FORTRAN V FLD function.) In BNF notation, this triplet is defined as:

⟨logical atom⟩ = . . . | (⟨logical sum⟩ , ⟨integral arithmetical sum⟩ , ⟨integral arithmetical sum⟩) | (⟨logical sum⟩ , ⟨integral arithmetical sum⟩) | . . . .

The elements of the triplet are:

a logical sum from which the extraction is to be performed

b integral arithmetical sum indicating the starting bit position

c integral arithmetical sum indicating the quantity of contiguous bits to be extracted

The third element defaults to a one. For example,

J = (I, 22, 3)

extracts the bits 22 23 and 24 from I and right justifies them. The equivalent FORTRAN coding is

DATA M3/Z7/

J = LAND(SHFTR(I, 7), M3)

N. Exclusive OR Operator

The exclusive OR operator X has been introduced at the top of the hierarchy of logical operators; i.e., over the inclusive OR operator V and the AND operator.. For example, the parentheses in the expression

(A.LE.B).(C.GT.D)X((E=F).(G$=H)V I)X(J.K.L)

are redundant; thus, the foregoing expression is equivalent to the expression

A.LE.B . C.GT.D X E=F.G$=H V I X J.K.L

O. Logical-Shift Operators

The logical-shift operators Q and R have been introduced over the top of the hierarchy of strictly logical operators. They are the logical-left-shift and logical-right-shift operators, respectively. Since these logical-shift operators are placed between dissimilar elements, they cannot be used more than once in a given logical sum. The OR operators X and V may be used left-recursively. In BNF notation, their comparative syntax is

⟨LS⟩ = ⟨LSX⟩ ⟨QR⟩ ⟨integer arithmetic sum⟩ | ⟨LSX⟩

⟨LSX⟩ = ⟨LSV⟩ X ⟨LSX⟩ | ⟨LSV⟩

⟨LSV⟩ = ⟨LP⟩ V ⟨LSV⟩ | ⟨LP⟩

.

.

.

⟨QR⟩ = Q | R

In the example

DATA M/Z380/

J = I . M R 7

we extract bits 22 23 and 24 from I and right justify them into J. The same result could be achieved by the triplet

```
J = (I 22 3)
```

The equivalent FORTRAN coding would be

```
DATA M/Z380/
J = SHFTR(LAND(I, M), 7)
```

and

```
DATA M3/Z7/
J = LAND(SHFTR(I, 7), M3)
```

respectively.

P. Generalized Limits for the DO-Index

In addition to permitting the usual integer or scalar integer variable as the two or three limits for the index in a DO-statement, a scalar arithmetical sum also is acceptable as any one or more of the DO-limits. For example, the one statement

```
DO I = I1+I2*I3 7*I4-I5 I6+1
```

is equivalent to the four statements

```
J1 = I1+I2*I3
J2 = 7*I4-I5
J3 = I6+1
DO I = J1 J2 J3
```

where the Js are dummy variables. Alternatively, a single integer array, with all of the subscripts except the first being specified integers or scalar integer variables which are constant within the DO-loop, may be specified as the set of values to be taken on by the DO-index. For example, if the array II has been declared

```
DIMENSION II(31 52 13)
```

then the one statement

```
DO I = II(I2-3*I12 2*I3)
```

is equivalent to the four statements

```
J2 = I2-3*I12
J3 = 2*I3
DO J1 = 1 31
I = II(J1 J2 J3)
```

The DO-array may *not* be employed in an implicit DO-statement, e.g., in an IO-statement.

Q. RETRIEVE

The built-in in-line integer replacement function of two variables RETRIEVE has been defined. This function uses a syntactical structure patterned after that of the ASSIGN function; namely,

RETRIEVE ⟨integer variable which reduces to a scalar⟩ = ⟨integer array⟩ (⟨integer variable which reduces to a scalar⟩)

For example, the declaration

```
DIMENSION K(100)
```

followed by the statement

```
RETRIEVE I = K(L)
```

is equivalent to the FORTRAN coding consisting of the same declaration

```
DIMENSION K(100)
```

followed by the five statements

```
    DO 1 I = 1,100
    IF(K(I).EQ.L) GO TO 2
1   CONTINUE
    I = 0
2   CONTINUE
```

The result is that I becomes the subscript whose corresponding element in the array K has the value of the argument L. If no element has the value L, then I is set to zero to indicate the failure to find a match.

If the array has more than one dimension, the additional subscripts may be specified immediately to the right of the second argument. These additional subscripts

are not involved in the retrieval process. In the foregoing example, let us assume that the array K had been declared

```
DIMENSION K(100 2 3)
```

and that the subscripts for the second and third dimensions are M2 and M3, respectively. Then the statement

```
RETRIEVE I = K(L M2 M3)
```

is equivalent to the same FORTRAN coding as in the original example, except that the IF-statement is replaced by

```
IF(K(I,M2,M3).EQ.L) GO TO 2
```

Thus, contrary to appearances, L is an argument but M2 and M3 are subscripts.

R. GOTO|SKIP

A syntactically new version of the GOTO has been introduced to facilitate the execution of a selected one out of several blocks of coding. The BNF definition of this version of the computed GOTO is

⟨GOTO⟩ = . . . |GO TO ⟨integer variable⟩, ⟨integer constant⟩| . . . .

The integer constant specifies how many alternative blocks of coding are expected to follow contiguously. The integer variable selects the individual block of coding to be executed immediately after the execution of the given GOTO.

The semantical requirement upon the alternatives is that the blocks of coding must follow the GOTO contiguously in sequence and that each block of coding be terminated by a SKIP statement. The amount of SKIPs (and hence of alternative blocks of coding) must equal that specified by the integer constant in the governing GOTO statement.

For example, the four-way branch coded in the TPL

```
GO TO I, 4

s1 *
SKIP

s2 *
SKIP

s3 *
SKIP

s4 *
SKIP

s5
```

is equivalent to the FORTRAN computed GOTO

```
GO TO (1, 2, 3, 4), I
```

immediately followed by the coding

```
1 s1 *
  GO TO 5
2 s2 *
  GO TO 5
3 s3 *
  GO TO 5
4 s4 *
5 s5
```

In each of the foregoing two cases, the statements s with subscripts are any executable statements. The asterisk * following an s indicates that one or more statements may be inserted at the indicated location in the coding.

These GOTO with their SKIPs may be semantically nested to any depth.

S. Four Lexical Structures for REAL Constants

Only four alternative lexical structures for a REAL constant are permitted; i.e.,

⟨sign⟩ ⟨one-or-more digits⟩ . ⟨one-or-more digits⟩

⟨sign⟩ ⟨one-or-two digits⟩|

⟨sign⟩ ⟨one-or-more digits⟩ . ⟨one-or-more digits⟩|

⟨one-or-more digits⟩ . ⟨one-or-more digits⟩

⟨sign⟩ ⟨one-or-two digits⟩|

⟨one-or-more digits⟩ . ⟨one-or-more digits⟩

T. Integer Constant | Variable Employed as Logical

Since any integer constant|variable may be employed as a logical constant|variable, there is no provision for the declaration of LOGICAL variables.

U. Single Character Symbols for Logical Operators

Single character symbols are used for certain of the logical operators:

TPL		FORTRAN	
.	=	.AND.	
V	=	.OR.	(inclusive)
|	=	.OR.	(inclusive)
X	=	.XOR.	(exclusive)
$	=	.NOT.	
¬	=	.NOT.	
=	=	.EQ.	
$=	=	.NE.	
¬=	=	.NE.	

These indicated FORTRAN style multiple-character symbols are *not* part of the TPL.

V. Abbreviations of Certain Keywords

The keywords

```
ANYHOW
DIMENSION
FUNCTION
OTHERWISE
RETRIEVE
RETURN
SUBROUTINE
```

optionally may be abbreviated in TPL to

```
ANY
DIM
FUN
OTH
RET
RET
SUB
```

respectively. For example, it is permissible to write

```
SUB A(B N C)
DIM B(N) C(N)
DO I = 1 N
C(I) = F(B(I))
DO END
RET
FUN F(X)
F = X*X
RET
```

W. A Subroutine Call Accomplished Without the Keyword CALL

A subroutine call is accomplished without the keyword CALL. Thus, for example,

```
SUBA(B C D E)
```

is equivalent to the FORTRAN

```
CALL SUBA(B,C,D,E)
```

The keyword CALL is *not* part of the TPL.

X. A Define-Before-Use Diagnostic

A semantical diagnostic system which reports any use of a variable before that variable has been defined and also reports any definitions of a given variable after it already has been defined once has been introduced. Therefore, the source coding has to be written in the order of execution; i.e., any unconditional, computed, or assigned GO TO has to point downwards, except if it is employed to close a DO-loop that has been coded explicitly.

The only syntactical effect is that the input and output variables of a subroutine (or function) call must be designated. The convention that all of the input variables are stated first while all of the output variables are stated last is adopted. A semicolon ; is required as a delimiter between these two sets of variables. For example, the coding in a routine

```
SUBA(B C ; D E)
```

calls the subroutine

```
SUBROUTINE SUBA(B C ; D E)
```

Here, we designated that the variables B and C are evaluated, prior to the call, in the higher level routine then used by the subroutine SUBA as input variables from which to compute the output variables D and E. These latter variables are passed up to the higher level routine for further use. If no semicolon is present, it is assumed that all the variables are input variables. It is a syntactical error to place the semicolon at the end of the calling sequence.

The semicolon syntactically is considered as another element in the calling sequence. Thus, if one insists in

inserting the optional delimiting commas, the foregoing two lines of coding would be

```
SUBA(B,C,;,D,E)
SUBROUTINE SUBA(B,C,;,D,E)
```

IV. An Example of the Usefulness of TPL in Telemetry

The following example shows how one might unpack successive values of the array IT, report the state of several two-bit logical variables, and print the values of several other integral variables. It is assumed that suitable declarations of the arrays would have DIMENSIONed them and set their values by means of, e.g., DATA statements.

The seven arrays contain the indicated information:

- IA list of the code numbers of the logical variables
- IB list of the code numbers of the integral variables
- ID list of the code numbers of all of the variables
- IO list of the subscripts of the array IT designating the element in which the given variable is located
- IP list of the applicable starting positions
- M list of the applicable masks
- IR list of the applicable required right shifts

The coding follows:

```
    DO ITT = 1, ITTMAX
    READ(5, 8) IT
 8  FORMAT(7Z8)
    IF(IT(...).mask)
    DO I=IA
    RET J = ID(I)
    K = IIT(IO(J) IP 2) + 1
    GO TO K, 4
    WRITE(6, 1) I
 1  FORMAT(1X, T16, I6, T28, 4HZERO)
    SKIP
    WRITE(6, 2) I
 2  FORMAT(1X, T16, I6, T28, 3HONE)
    SKIP
    WRITE(6, 3) I
 3  FORMAT(1X, T16, I6, T28, 3HTWO)
    SKIP
    WRITE(6, 4) I
 4  FORMAT(1X, T16, I6, T28, 5HTHREE)
    SKIP
    WRITE(6, 5)
 5  FORMAT(13H+THE VALUE OF, 10X,
          3HWAS)
    DO END
    DO I=IB
    RET J = IB(I)
    K = IT(10(J)).M(J) R IR(J)
    WRITE(6, 6) I, K
 6  FORMAT(15H THE VALUE OF , I6,
          7H WAS    , I12)
    DO END
    DO END OTHERWISE
    WRITE(6, 9) ITT
 9  FORMAT(33H1REQUESTED TERMINATION
          ON ITT = , I6)
    ANYHOW
    IF($ITT) WRITE(6, 10)
10  FORMAT(22H1EXHAUSTED TERMINATION)
```

V. Remarks

Some of the features of the TPL were already present in the spacecraft-simulation procedural language. These features are described in greater detail in the original articles (Refs. 1 and 2).

References

1. Ścibor-Marchocki, R. I., "A Spacecraft-Simulation Problem-Oriented Language," in *The Deep Space Network*, Space Programs Summary 37-58, Vol. II, pp. 87–96. Jet Propulsion Laboratory, Pasadena, Calif., July 31, 1969.
2. Ścibor-Marchocki, R. I., "Additional Features of the Spacecraft-Simulation Problem-Oriented Language," in *The Deep Space Network*, Space Programs Summary 37-62, Vol. II, pp. 99–107. Jet Propulsion Laboratory, Pasadena, Calif., Mar. 31, 1970.